

Complexity IBC028, Lecture 7

H. Geuvers

Institute for Computing and Information Sciences
Radboud University Nijmegen

Version: spring 2024



Outline

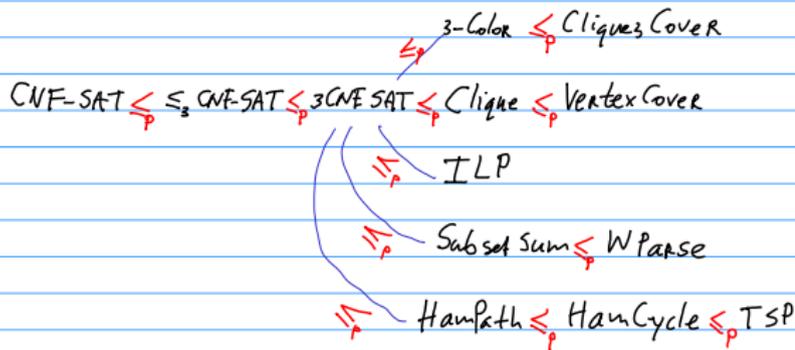
SAT is **NP**-complete

Course Overview





Overview of NP-complete problems



Recap on NP-complete problems I

- **NP** problems:

NP :=

$$\{A \subseteq \{0, 1\}^* \mid \exists f, f \text{ polynomial,} \\ x \in A \iff \exists y \in \{0, 1\}^* (|y| \text{ polynomial in } |x| \wedge f(x, y) = 1)\}$$

- We know $\mathbf{P} \subseteq \mathbf{NP}$.
- A big open question is whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.
- **NP**-hard problems:

$$\mathbf{NPH} := \{A \mid \forall X \in \mathbf{NP} (X \leq_P A)\}$$

- **NP**-complete problems:

$$\mathbf{NPC} := \mathbf{NPH} \cap \mathbf{NP}$$

- If one **NPH** problem is in **P**, then all **NP** problems are in **P**.
(So **NP**-hard problems are likely to be the “hardest **NP** problems.”)

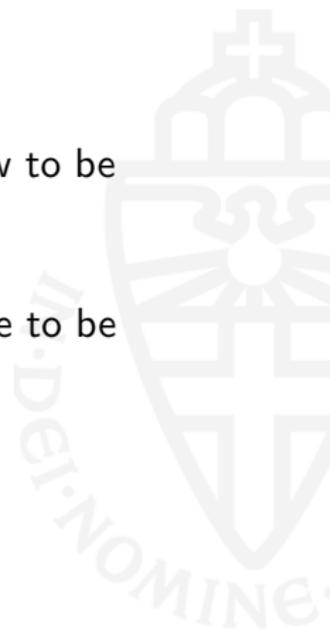
Recap on **NP**-complete problems II

Method for showing that problem A is **NP**-complete:

- Show that $A \in \mathbf{NP}$. (Usually quite easy.)
- Show that $B \leq_P A$ for a problem B that we know to be **NP**-hard (because then A is **NP**-hard as well).

So: we have to start from some problem that we prove to be **NP**-hard!

That problem is SAT (or CNF-SAT)



The Cook Levin Theorem

THEOREM, Cook - Levin, 1971, 1973

SAT is **NP**-complete

One often follows the proof of Karp 1972, proving that CNF-SAT is **NP**-complete

PROOF

- SAT \in **NP**: for φ a boolean formula, the certificate is the satisfying assignment v ; v is polynomial in $|\varphi|$ and checking $v(\varphi) = 1$ is also polynomial.
- SAT \in **NPH**.
This is the hard part...and the main content of the Cook-Levin theorem.

SAT is NP-hard

We need to prove that for all $A \in \mathbf{NP}$,

$$A \leq_P \text{SAT}.$$

That is: for every $A \in \mathbf{NP}$ we should find a polynomial function h_A such that

$$\forall x (x \in A \iff h_A(x) \in \text{SAT}).$$

But A can be anything: Ham-Cycle, ILP, VertexCover,
and A can be about graphs, integers, points in \mathbb{R}^2 ,

What do we know about A ?

$A \in \mathbf{NP}$, so there is a **polynomial** function f such that

$$x \in A \iff \exists y \in \{0, 1\}^* (|y| \text{ polynomial in } |x| \wedge f(x, y) = 1).$$

We will construct a function h that mimicks the function f as a SAT-formula.

SAT is NP-hard

PROOF

For every $A \in \mathbf{NP}$ we should find a polynomial h_A such that

$$\forall x (x \in A \iff h_A(x) \in \text{SAT}).$$

For $A \in \mathbf{NP}$ there is a polynomial f such that

$$x \in A \iff \exists y \in \{0, 1\}^* (|y| \text{ polynomial in } |x| \wedge f(x, y) = 1)$$

The h_A we construct will mimick f .

- We use **Turing Machines** to talk about this f :
- f is given by a polynomial time Turing Machine M .
- h_A will mimick the polynomial time Turing Machine M that decides A .

Encoding a Turing Machine as a boolean formula (I)

f is given by a polynomial time Turing Machine $M = (Q, \Sigma, \delta)$ and we have

$$f(x, y) = 1 \iff M \text{ halts in state } q_F \text{ on input } (x, y).$$

We will encode the operation of M on (x, y) as a boolean formula.

- We take $\Sigma = \{0, 1, \sqcup, \dots\}$.
- For readability, we also use \rightarrow as a boolean connective.
- We use $v(p) \in \{\text{true}, \text{false}\}$ to distinguish the satisfiability problem we construct from the 0 and 1 as tape content.
- A configuration of M is given by: a state q and tape content $a_1 \dots a_k a_{k+1} \dots a_n$ with q reading a_k . We encode this by

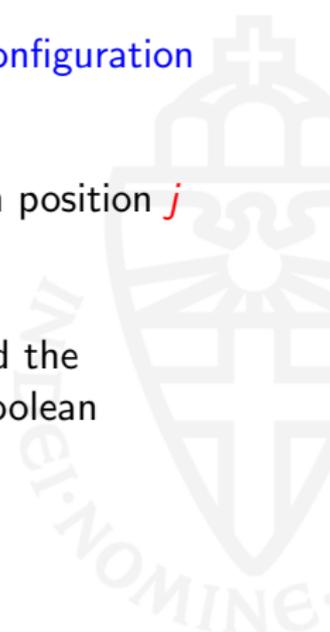
$$a_1 \dots a_k q a_{k+1} \dots a_n \in (Q \cup \Sigma)^*$$

Encoding a Turing Machine as a boolean formula (II)

- We introduce boolean variables to describe the **configuration of M after i steps**. Intended meaning:

$p_{i,j,a} = \text{true} \iff$ after i steps, there is an a on position j

- We will encode the intended meaning of $p_{i,j,a}$ and the operations of M by writing a (vast) number of boolean formulas.



Encoding a Turing Machine as a boolean formula (III)

The boolean variables $p_{i,j,a}$ should together represent the state of the Machine M in a computation:

$$a_1 \dots a_k q a_{k+1} \dots a_n \in (Q \cup \Sigma)^*$$

But the tape is infinite...??

We know:

$$x \in A \iff \exists y \in \{0, 1\}^* (|y| \text{ polynomial in } |x| \wedge$$

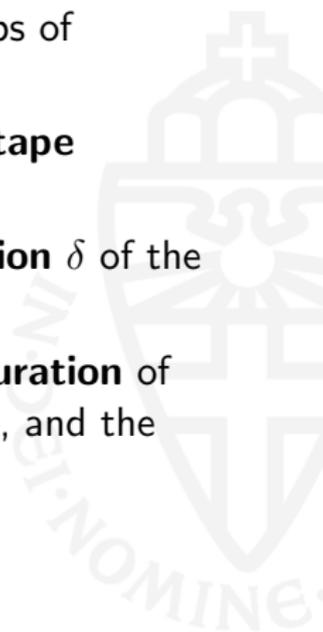
M halts in polynomial time in state q_F on input (x, y)).

- $|y|$ is polynomial in $|x|$, so $|y| \leq c|x|^k$ (for some k and c).
- M is polynomial in $|x| + |y|$, so there are ℓ and d such that
 - computation of M on (x, y) takes $\leq d(|x| + |y|)^\ell$ steps,
 - so computation of M on (x, y) uses $\leq d(|x| + |y|)^\ell$ symbols on tape.
- So the number of boolean variables is bound by $(d(|x| + c|x|^k)^\ell \times (d(|x| + c|x|^k)^\ell \times (|\Sigma| + |Q|))$, so bound by a polynomial $P(|x|)$.

Encoding a Turing Machine as a boolean formula (IV)

Using the boolean variables $p_{i,j,a}$ we define three groups of formulas.

- 1 Boolean formulas that describe properties that a **tape configuration** should obey,
- 2 Boolean formulas describing the **transition function** δ of the Turing Machine,
- 3 Boolean formulas that describe the **initial configuration** of the Turing Machine, with the input x on the tape, and the final **accepting configuration**.



Encoding a Turing Machine as a boolean formula (V)

(1) Boolean formulas to describe **tape configurations**

$$\bigwedge_{i,j} \left(\left(\bigvee_{a \in \Sigma \cup Q} p_{i,j,a} \right) \wedge \bigwedge_{a,b \in \Sigma \cup Q, a \neq b} (\neg p_{i,j,a} \vee \neg p_{i,j,b}) \right)$$

- On every i (every time step) each j (every tape location) holds an $a \in \Sigma \cup Q$,
- On every i (every time step) each j (every tape location) holds at most one $a \in \Sigma \cup Q$.

Note that both i and j are bound by $P(|x|)$, so the size of this formula is polynomial in $|x|$.

Encoding a Turing Machine as a boolean formula (VI)

(2) Boolean formulas describing the **transition function** δ .

Suppose that we have $\delta(q, a) = (q', b, R)$.

We add, for every i, j and every $c \in \Sigma$ the formula

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,c}) \rightarrow (p_{i+1,j,b} \wedge p_{i+1,j+1,c} \wedge p_{i+1,j+2,q'})$$

The rest of the tape remains intact so we add, for every $d \in \Sigma$, and for every $k < j$ and every $k > j + 2$ the formula

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,c}) \rightarrow (p_{i+1,k,d} \leftrightarrow p_{i,k,d})$$

Note that again, i, j and k are bound by $P(|x|)$, so the size of this formula is polynomial in $|x|$.

This is repeated for all transition steps of δ .

Encoding a Turing Machine as a boolean formula (VII)

(3) Boolean formulas describing the initial configuration of the Turing Machine with input x (and certificate y “to be guessed”), and the accepting condition.

- $p_{0,1,q_0}$
- $p_{0,j+1,0}$ for all j -positions in x for which $x_j = 0$
- $p_{0,j+1,1}$ for all j -positions in x for which $x_j = 1$
- $p_{0,|x|+2,e}$ marking the end of input x , for marking symbol e
- $p_{0,|x|+2+j,0} \vee p_{0,|x|+2+j,1}$ for all j -positions in y , which should be either 0 or 1
- $p_{0,j,\sqcup}$ for all other tape positions, for the “blank” symbol \sqcup .
- $\bigvee_{i,j} p_{i,j,q_F}$ describing that M has reached the final state q_F .

Note that again, i, j are bound by $P(|x|)$, so the size of this formula is polynomial in $|x|$.

Encoding a Turing Machine as a boolean formula (VIII)

Given Turing Machine M (that implements algorithm f), and input x , we denote by $h_M(x)$ the Boolean formula that is the conjunction of all the formulas that we have just described.

We have the following:

$$h_M(x) \in \text{SAT}$$

\iff the $p_{0,j,a}$ describe a valid initial configuration with x as input and some choice for y
and $\forall i > 0$, the $p_{i,j,a}$ describe a configuration of M after i steps

and $\bigvee_{i,j} p_{i,j,q_F} = \text{true}$

(at a certain point we arrive at state q_F)

$\iff \exists y (|y| \text{ poly. in } |x| \wedge M \text{ with input } (x, y) \text{ halts in } q_F)$

$\iff \exists y (|y| \text{ poly. in } |x| \wedge f(x, y) = 1).$

So: For every $A \in \mathbf{NP}$ ($A \leq_P \text{SAT}$).

So: $\text{SAT} \in \mathbf{NPH}$ and so $\text{SAT} \in \mathbf{NPC}$.



CNF-SAT is NP-complete

The construction of f in the proof can be adapted a bit so that $f(x)$ is a CNF-formula.

Steps (1) and (3) already create a CNF. For Step (2):

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,c}) \rightarrow (p_{i+1,j,b} \wedge p_{i+1,j+1,c} \wedge p_{i+1,j+2,q'})$$

is equivalent to the three clauses

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,c} \vee p_{i+1,j,b}$$

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,c} \vee p_{i+1,j+1,c}$$

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,c} \vee p_{i+1,j+2,q'}$$

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,c}) \rightarrow (p_{i+1,k,d} \leftrightarrow p_{i,k,d})$$

is equivalent to the two clauses

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,c} \vee p_{i+1,k,d} \vee \neg p_{i,k,d}$$

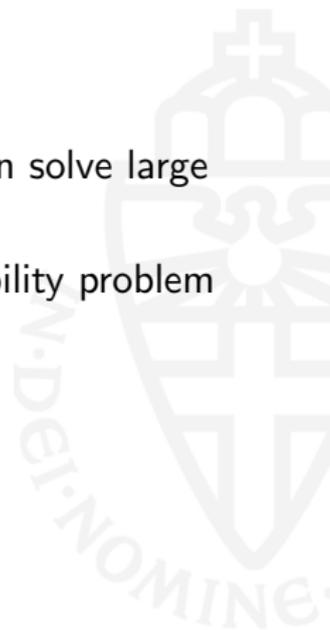
$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,c} \vee \neg p_{i+1,k,d} \vee p_{i,k,d}$$

So, for every $A \in \mathbf{NP}$ ($A \leq_P \text{CNF-SAT}$) and so: $\text{CNF-SAT} \in \mathbf{NPH}$.

Why SAT is important

SAT is **NP**-complete, but

- nevertheless there are very powerful tools that can solve large SAT problems (and even a bit more) very quickly
- many decision problems can be cast as a satisfiability problem





Example: Bounded Model Checking

Consider the following algorithm that sorts a triple of booleans.

```
if  $a_1 > a_2$  then swap( $a_1, a_2$ );  
if  $a_2 > a_3$  then swap( $a_2, a_3$ );  
if  $a_1 > a_2$  then swap( $a_1, a_2$ )
```

Question: is this a correct sorting algorithm?

Introduce variables $a_{i,j}$ as values of a_j after i steps ($i = 0, 1, 2, 3$) and introduce boolean formulas to denote the steps in the algorithm. For the first step:

$$\begin{aligned}(a_{0,1} \wedge \neg a_{0,2}) &\rightarrow (a_{1,1} \leftrightarrow a_{0,2} \wedge a_{1,2} \leftrightarrow a_{0,1} \wedge a_{1,3} \leftrightarrow a_{0,3}) \\ \neg(a_{0,1} \wedge \neg a_{0,2}) &\rightarrow (a_{1,1} \leftrightarrow a_{0,1} \wedge a_{1,2} \leftrightarrow a_{0,2} \wedge a_{1,3} \leftrightarrow a_{0,3})\end{aligned}$$

Add a boolean formula that states that the algorithm is incorrect:

$$(a_{3,1} \wedge \neg a_{3,2}) \vee (a_{3,2} \wedge \neg a_{3,3})$$

The conjunction of these formulas is **not satisfiable**, so the algorithm is correct.

Course overview(I)

1 Divide and Conquer algorithms

If #steps on input of size n is $T(n)$, we have

$$T(n) = \sum_{\text{some } k, k < n} T(k) + f(n)$$

2 How to derive a $g(n)$ such that $T(n) = \mathcal{O}(g(n))$? (or $\Omega(g(n))$, $\Theta(g(n))$)

- Substitution method
- Recursion tree method
- Master Theorem method, for $T(n) = aT(\frac{n}{b}) + f(n)$.

3 Example algorithms:

- Karatsuba multiplication of numbers: $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$.
- The median of a list of numbers of length n , in $\Theta(n)$.
- Matrix multiplication (and inversion): $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.8})$.

Course overview(II)

4 P and NP

P :=

$$\{A \subseteq \{0, 1\}^* \mid \exists f, f \text{ polynomial}, w \in A \iff f(w) = 1\}$$

NP :=

$$\{A \subseteq \{0, 1\}^* \mid \exists f, f \text{ polynomial}, \\ w \in A \iff \exists y \in \{0, 1\}^* (|y| \text{ polynomial in } |w| \wedge f(w, y) = 1)\}$$

5 NP-hard, NP-complete and reductions.

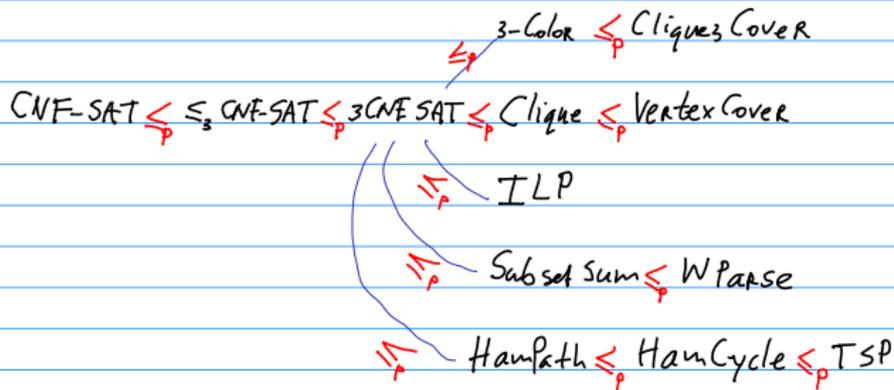
- **NPH** := $\{A \mid \forall X \in \mathbf{NP} (X \leq_P A)\}$
- **NPC** := $\mathbf{NP} \cap \mathbf{NPH}$
- $A_1 \leq_P A_2$ if
 \exists polynomial $f : \{0, 1\}^* \rightarrow \{0, 1\}^* (x \in A_1 \iff f(x) \in A_2)$
- (Theorem) If and $A \in \mathbf{NPH}$ and $A \leq_P B$, then $B \in \mathbf{NPH}$.
- (Theorem) $\text{SAT} \in \mathbf{NPC}$



Course overview(III)

6

- Whole list of **NPC**-problems:



Course overview(IV)

7 PSPACE

- Definition of **PSPACE**-problem, **PSPACE**-complete
- QBF and variants are **PSPACE**-complete

